Advances in Operating Systems CS60038

Term Project

Group - 13 Networking support in OS

Roll Number	Name	Research Paper
20CS10085	Pranav Mehrotra	**Eiffel**: efficient and flexible software packet scheduling, NSDI 2019
20CS30065	Saransh Sharma	**Shenango**: achieving high CPU efficiency for latency-sensitive datacenter workloads, NSDI 2019
20CS10088	Shah Dhruv Rajendrabhai	**TCP ≈ RDMA**: CPU-efficient remote storage access with i10, NSDI 2020
20CS10077	Vivek Jaiswal	**Loom**: flexible and efficient NIC packet scheduling, NSDI 2019

Eiffel: efficient and flexible software packet scheduling, NSDI 2019

a) **General Introduction:**

   i) **Comparison of Software and Hardware Solutions:** In the landscape of modern networks, software-based packet scheduling stands out for its advantages, including shorter development cycles and greater deployment flexibility, compared to its hardware counterparts.

   ii) **Research Focus:** The primary focus is on designing and implementing software-based packet scheduling, intending to overcome existing limitations observed in current solutions.

b) **Motivation:**

   i) **Need for Improvement in Software Schedulers:** Recognizing the inefficiencies and limitations of current software packet schedulers, there is a pressing need to enhance their performance while preserving flexibility substantially.

   ii) **Advantages of Software Solutions:** The motivation stems from acknowledging the inherent benefits of software solutions, including their adaptability to diverse network needs and the potential to serve as replacements or precursors for hardware schedulers.

   iii) **Increasing Demand for Programmability:** With the rising complexity of network policies and the deployment of schedulers at multiple points in a packet's path, there is an escalating demand for programmable schedulers that can efficiently handle high packet rates.

   iv) **Addressing Inefficiency Challenges:** Current software schedulers face challenges in efficiently handling packets at high rates, often resulting in significant CPU overhead. The motivation lies in devising solutions to address these inefficiencies and enhance overall system performance.

   v) **Efficient Data Structures:** Eiffel aims to develop data structures for any scheduling algorithm with O(1) processing overhead per packet, leveraging integer priority queues.

   vi) **Programmable Abstraction:** Eiffel aims to provide a fully expressive scheduler programming abstraction by extending the PIFO model, allowing operators to specify policies based on their requirements and available resources.

c) **Past Works:**

   i) Existing software packet schedulers have limitations in efficiency and flexibility:

      1) FQ/Pacing qdisc uses inefficient RB-trees resulting in high overhead.

      2) Carousel provides efficient shaping but supports only non-work-conserving scheduling.

      3) OpenQueue allows programmable scheduling but uses inefficient priority queue building blocks.

      4) QFQ approximates fair queuing efficiently but is tailored only for specific algorithms.

   ii) Hardware solutions also have constraints:

      1) PIFO enables programmable scheduling with efficient primitives but has limited capacity.

      2) PIFO lacks support for per-flow scheduling.

   iii) There is a need for a software packet scheduler that:

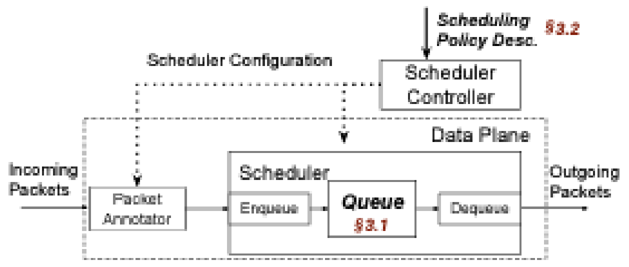
      1) Combines efficiency through novel data structures like integer priority queues.

      2) Provides flexibility via an expressive programming model.

    3) Can capture diverse scheduling policies.

d) Design Proposal of Eiffel:

- **i)** Main goal is to design an efficient and flexible software packet scheduler that can be deployed in kernel space and userspace.
- **ii)** Proposes a modular architecture with four main components:
    - **1)** **Packet Annotator:** Sets input priority or rank for incoming packets
    - **2)** **Enqueue Module:** Calculates rank for incoming packets based on policy
    - **3)** **Priority Queue:** Holds packets sorted by rank
    - **4)** **Dequeue Module:** Reranks packets in queue on dequeue based on policy
- **iii)** Leverages integer priority queues to achieve O(1) overhead per packet for enqueue and dequeue operations. Proposes two integer priority queue designs:
    - **1)** **Circular Find First Set (FFS)-based Queue (cFFS):**
        - **(a)** Extends priority queues based on Find First Set (FFS) CPU instruction.
        - **(b)** FFS can find the leftmost set bit in constant time.
        - **(c)** Works for priority levels less than FFS word size.
        - **(d)** Uses a hierarchy of bitmaps for a larger number of priority levels.
        - **(e)** Introduces circular buffering technique to handle a moving range of priorities.
    - **2)** **Approximate Gradient Queue:**
        - **(a)** Uses algebraic approach to approximate highest priority index.
        - **(b)** Models queue occupancy as a curvature function.
        - **(c)** Critical point of curvature indicates highest priority element.
        - **(d)** Allows approximation to reduce steps to find the highest priority.
        - **(e)** Trades off accuracy for improved efficiency in some cases.
        - **(f)** Extends Push In First Out (PIFO) scheduler programming model to add support for a wider range of policies:
- **iv)** **Adds per-flow packet scheduling primitive:**
    - **1)** Ranks and schedules packets grouped by flows.
    - **2)** Incoming packet changes rank of all packets in the same flow.
- **v)** **Adds on-dequeue ranking primitive:**
    - **1)** Allows reranking packets on dequeue based on policy.
    - **2)** Enables arbitrary traffic shaping through a separate shaper module.
    - **3)** Decouples shaping from scheduling hierarchy.



e) Implementation of Eiffel:

- **i)** Implemented in kernel space as a queuing discipline (qdisc) module in Linux.
- **ii)** Also implemented in userspace in the Berkeley Extensible Software Switch (BESS).
- **iii) The kernel implementation:**
  - **1)** Integrates with existing qdisc infrastructure in the Linux kernel.
  - **2)** Uses proposed cFFS or gradient queue for packet queue data structure.
  - **3)** Optimizes finding minimum deadline packets to set dequeue timer efficiently.
- **iv) The BESS implementation:**
  - **1)** Leverages BESS's pipeline processing model.
  - **2)** Uses optimizations like per-flow batching, output batching, and max packets per queue.
  - **3)** Achieves high throughput via busy polling on multiple cores.
  - **4)** Both implementations exploit the O(1) priority queues to minimize packet processing overhead.

f) Advantages of Eiffel:

- **i)** Achieves constant O(1) enqueue/dequeue overhead per packet independent of the number of flows.
- **ii)** Enables line-rate packet scheduling at scale regarding the number of flows.
- **iii)** Supports a wide range of scheduling policies with a flexible programming model:
  - **1)** Per-flow scheduling policies like Weighted Fair Queueing
  - **2)** Deadline-based policies like Earliest Deadline First
  - **3)** Time-based policies like packet pacing/shaping
  - **4)** Priority-based policies like Strict Priority Queueing
  - **5)** Load-based policies like Least Slack Time First
  - **6)** Examples implemented: pFabric, hClock, Carousel
- **iv)** Ease of implementation in kernel and userspace environments.
- **v)** Integer priority queues provide significant efficiency gains over traditional priority queues.

vi) Modular architecture allows custom scheduling policies to be easily expressed.

g) **Evaluation Methodology:**

i) **Kernel implementation:**

1) Tested as qdisc module in Linux kernel v4.10

2) Evaluated CPU utilization for pacing and shaping scenarios

3) Generated traffic using neper tool, sockets modified to avoid flow tracking

ii) **BESS implementation:**

1) Implemented as modular pipeline in BESS software switch

2) Evaluated maximum achievable load for pFabric scheduling

3) Used simple flow generator, 1500B packets, single core

iii) **Baselines:**

1) FQ/Pacing qdisc and Carousel (kernel)

2) Binary heap priority queue pFabric (BESS)

h) **Results:**

i) 14x lower CPU utilization than FQ/Pacing for kernel pacing scenario with 20Gbps traffic rate.

ii) 3x lower CPU overhead compared to Carousel for traffic shaping use case in kernel.

iii) 5x higher achievable load than binary heap implementation of pFabric in BESS for given packet rate.

iv) Microbenchmarks demonstrate Eiffel sustaining line rate for 5x more flows than binary heap.

v) Approximate priority queue provides up to 9% performance gain over cFFS queue in high load scenarios.

vi) Evaluations across real-world platforms like Linux and BESS demonstrate consistent gains.

i) **Future Enhancements:**

i) Leverage Eiffel's integer priority queue concepts to optimize hardware packet scheduler designs.

ii) Develop advanced techniques like dynamically configurable non-uniform bucket sizes.

iii) Explore auto-tuning and machine learning-based optimization of Eiffel's parameters.

iv) Analyze Eiffel benefits for emerging workloads like AI/ML training and inference pipelines.

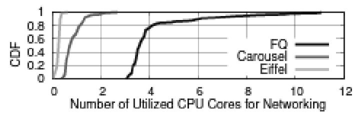
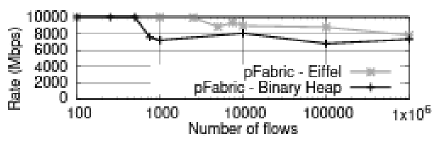
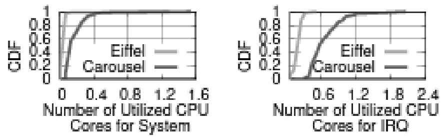


Figure 8: A comparison between the CPU overhead of the networking stack using FQ/pacing, Carousel, and Eiffel.





Shenango - Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads

a) **General Introduction**

   i) Modern datacenter applications demand unprecedented performance metrics, including microsecond-scale tail latency, high throughput, and optimal CPU efficiency.

   ii) Challenge lies in effectively balancing these requirements amid the inherent workload variability experienced across multiple timescales.

b) **Motivation**

   i) Challenges with Kernel Bypass Networking

      1) Conventional kernel networking stacks like Linux, are constrained by millisecond-scale tail latency due to kernel overheads.

      2) Kernel bypass approaches, such as ZygOS, are inefficient in dedicating cores and wasting CPU cycles during non-peak loads.

   ii) Datacenter Workload Imperatives

      1) The imperative requirements of modern datacenter workloads, includes microsecond-scale tail latency, high throughput, and optimal CPU efficiency.

      2) Workload variability and its impact on performance objectives poses intricate challenges.

   iii) Systemic Challenges

      1) Existing systems, such as those built on Linux or leveraging user-level schedulers like Arachne, struggles in accommodating workload variability.

      2) Need for agile core reallocation to effectively multiplex CPUs at the granularity of individual requests, addressing a crucial gap in current approaches.

c) **Prior Works**

   i) Kernel Threading Focus

      1) Prior works, including Arachne and Linux, primarily address kernel threading overheads.

      2) These solutions operate at millisecond granularity, limiting their ability to handle microsecond-scale latency requirements seen in modern datacenter workloads.

   ii) Kernel Bypass Techniques

      1) Kernel bypass techniques like ZygOS and IX reduce networking overheads through userspace network stacks.

      2) Despite achieving lower latency, these systems lack the capability for microsecond-scale core multiplexing, compromising efficiency during workload fluctuations.

   iii) Dynamic Resource Allocators

      1) Dynamic resource allocators in systems like IX, Arachne, and Linux optimize core allocation based on aggregate metrics over long timescales.

      2) Ineffective for microsecond decision-making, these allocators struggle to swiftly adapt to rapid changes in workload, hindering their responsiveness.

iv) **Tail Latency Improvement Efforts**

1) Various efforts focus on improving tail latency through load balancing, isolation, and work stealing techniques.

2) While effective in enhancing tail latency, these systems inherit limitations from underlying resource allocation mechanisms that operate at coarse granularity, impacting their ability to handle microsecond-scale adjustments.

d) **Design Proposal:**

i) **Two-Level Architecture:**

1) Introduction of a novel two-level architecture for efficient resource multiplexing.

2) Per-application runtimes linked to the Shenango runtime library.

3) Dedicated IOKernel managing physical resources and core allocation.

ii) **Kernel-Bypass Abstractions:**

1) Shenango runtime library provides high-level OS abstractions for applications.

2) Implementation of efficient kernel-bypass equivalents for threading, synchronization, and sockets.

3) Initialization of kernel threads with core-local runqueues for precise scheduling.

iii) **Dynamic Core Allocation:**

1) IOKernel employs a 5µs congestion detection algorithm for swift decision-making.

2) Coordinated core allocation in response to runtime thread and packet queue inspections.

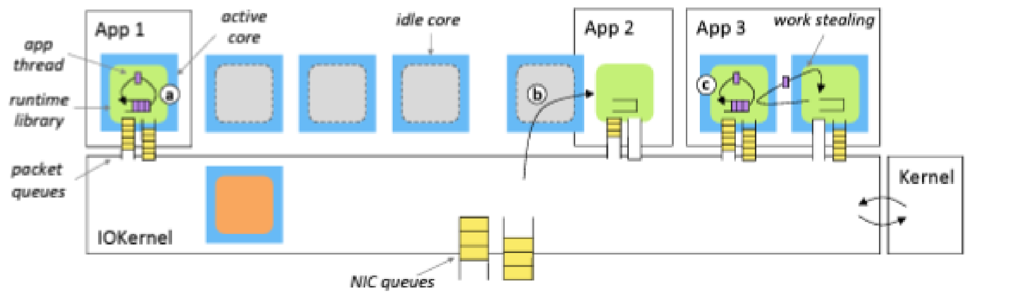
3) Priority given to hyperthread siblings and recently freed cores for enhanced cache efficiency.

iv) **Fine-Grained Multiplexing:**

1) Runtimes exclusively run on allocated cores, voluntarily yielding when idle.

2) Busy-spinning IOKernel facilitates coordinated core allocation and packet I/O.

3) Fine-grained core multiplexing balances efficiency and low latency, adapting to workload variability.



e) **Design Details:**

i) Shenango Runtime

1) Userspace library linked to applications for kernel-bypass abstractions.
2) Implements OS functionalities, including threads, mutexes, and sockets.
3) Maintains kernel threads with core-local runqueues for efficient scheduling.

ii) IOKernel

1) Centralized and privileged controller operating on a dedicated core.
2) Busy-spinning loop for NIC queue, runtime state, and transmission queue monitoring.
3) Implements direct network packet steering based on IP/MAC addresses and enqueues to per-core ingress queues.

iii) Congestion Detection Algorithm

1) Novel 5µs algorithm inspects runtime thread and packet queues for incipient overload signs.
2) Triggers microsecond-scale core allocation decisions.
3) Utilizes shared memory state for efficient runtime and IOKernel communication.

iv) Voluntary Yielding and Preemption

1) Runtimes yield cores voluntarily when idle to optimize resource utilization.
2) IOKernel can forcibly preempt non-guaranteed cores for enhanced control.
3) Contributes to fine-grained and coordinated core multiplexing.

f) Implementation:

i) Technology Stack

1) Implemented in C and C++, with additional Rust support.
2) Compatible with Linux, leveraging existing kernel APIs for seamless integration.
3) IOKernel utilizes Intel DPDK for userspace NIC access, optimizing network performance.

ii) Inter-Process Coordination

1) Shared memory regions utilized for efficient packet queues, command passing, and state sharing.
2) Affinity APIs employed to pin threads and bind runtime cores.
3) Inter-process signaling via signals and eventfds for thread parking/unparking.

iii) Scalability Considerations

1) Designed for a single socket but extendable for NUMA scaling using multiple IOKernel instances.
2) Shared memory and signaling facilitate lightweight inter-process coordination.
3) Affinity APIs ensure optimal core binding for improved scalability.

g) Benefits of Shenango:

i) **Microsecond-Scale Tail Latency:** Shenango achieves microsecond-scale tail latency, responding to sudden bursts in load with minimal latency increase. This capability ensures superior responsiveness for time-sensitive datacenter applications.

ii) **High Throughput with Efficient CPU Utilization:** Shenango demonstrates high throughput, handling over five million requests per second for memcached, while efficiently utilizing CPU resources. Its fine-grained core multiplexing balances workload variability, ensuring optimal performance even under diverse conditions.

iii) **Dynamic Core Allocation for Adaptive Workload Handling:** The innovative two-level architecture of Shenango, with per-application runtimes and a dedicated IOKernel, allows dynamic core allocation at a 5µs granularity. This ensures swift adaptation to workload changes, enabling efficient resource utilization and low tail latency.

iv) **Efficient Thread Scheduling:** Shenango's thread scheduling outperforms existing systems, providing low-latency and atomic-free wakeups. Preallocated stacks and careful attention to thread management contribute to enhanced efficiency in handling high-level programming abstractions.

v) **Resilience to Load Bursts:** Shenango demonstrates remarkable resilience to sudden load shifts, reacting quickly to maintain low tail latency even under extreme transitions. This adaptability ensures stable and responsive performance in dynamic datacenter environments.

h) Evaluation Methodology:

i) **Hardware Configuration:**

1) Employed a dual-socket server equipped with 12-core Intel Xeon E5-2650v4 CPUs (2.20GHz).

2) Total RAM of 64GB and a high-speed Intel 82599ES NIC with 10Gbits/s capacity.

3) Enabled hyper-threads and evaluated only the first socket for focused analysis.

4) Configured the server to minimize jitter by disabling TurboBoost, C-states, and CPU frequency scaling.

ii) **Load Generation Setup:**

1) Generated load from six additional quad-core machines connected through a Mellanox SX1024 switch.

2) Utilized Mellanox ConnectX-3Pro NICs for efficient communication with the main server.

3) Load patterns simulated using open-loop Poisson processes to model packet arrivals.

iii) **Systems Under Evaluation:**

1) Compared Shenango's performance against Arachne, ZygOS, and Linux.

2) Arachne, a user-level threading system, with core allocation adjustments over millisecond timescales.

3) ZygOS, a kernel-bypass network stack, incorporating fine-grained load balancing but lacking thread support.

4) Linux, widely deployed but constrained by kernel overheads.

i) Results:

i) **Memcached Performance**

1) Throughput: Shenango achieves over five million requests per second while maintaining a median response time of 37µs and a 99.9th percentile response time of 93µs. ZygOS, despite busy polling, maintains similar response times only up to four million requests per second.

2) Efficiency: Shenango outperforms others in batch processing efficiency despite reserving two hyperthreads for the IOKernel.

ii) **Spin-Server Handling Service-Time Variability**

1) Latency: Shenango demonstrates slightly higher throughput than ZygOS for the spin-server with three service-time distributions, maintaining similar tail latency. Arachne exhibits higher throughput than Linux and performs well, although with slightly higher tail latency at lower loads.

2) Batch Throughput: Shenango maintains batch throughput efficiently even with two hyperthreads reserved for the IOKernel.

iii) **Resilience to Load Bursts**

1) Tail Latency Under Load Shifts: Shenango reacts swiftly to load shifts, handling an extreme transition from 100,000 to 5 million requests per second with almost no additional tail latency, showcasing its adaptability and quick response.

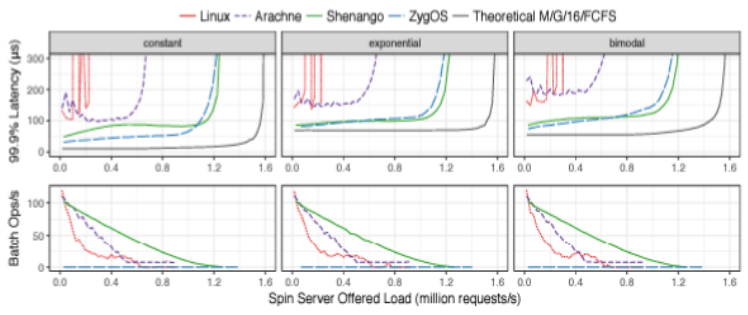
j) **Future Enhancements:**

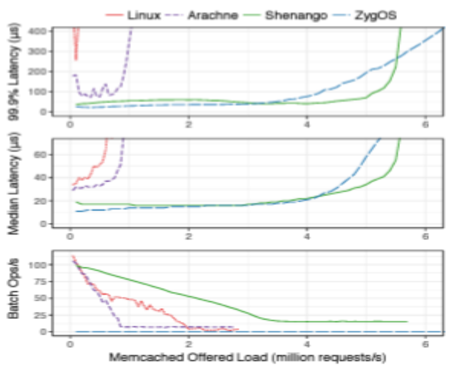
i) **Optimizing IOKernel Algorithms:** Further refinement of the congestion detection algorithm within the IOKernel could enhance its ability to react to application overload. Fine-tuning this algorithm may lead to even more precise and rapid core allocation adjustments, ensuring optimal system performance under varying conditions.

ii) **Dynamic Core Allocation Policies:** Exploring dynamic policies for core allocation could be beneficial. Adapting the core allocation strategy based on real-time workload characteristics and system demands might further improve Shenango's responsiveness and efficiency, especially in scenarios with evolving workloads.

iii) **Enhanced Network Stack Integration:** Continuous improvements in network stack integration could contribute to minimizing network latencies further. Exploring advanced techniques for packet processing and network communication within the IOKernel may unlock additional optimizations, reinforcing Shenango's position as a high-performance system.

iv) **Scalability Across NUMA Domains:** Investigating strategies to extend Shenango's design for improved scalability across Non-Uniform Memory Access (NUMA) domains could be valuable. Adapting the system to leverage multiple IOKernel instances for NUMA scaling might enhance its applicability in large-scale, distributed environments.





TCP ≈ RDMA: CPU-efficient Remote Storage Access with i10:

a) **General Introduction**

In this research endeavor, the authors introduce "i10," an innovative remote storage stack seamlessly integrated into the kernel. This development addresses the need for structural modifications to achieve efficient remote storage access, a demand accentuated by substantial enhancements in internet and storage hardware speeds. The investigation sheds light on the impractical CPU overheads associated with conventional solutions for remote storage access.

b) **Motivation**

i) **Advancements in Network and Storage Hardware**

Surge in Bandwidth and NVMe Adoption: The primary motivation for i10 stems from the remarkable progress in network and storage hardware, with network access bandwidth escalating from 1Gbps to 40Gbps and, in some cases, reaching 100Gbps. Simultaneously, the widespread adoption of NVMe storage devices, capable of delivering over a million IOPS, reshapes the storage landscape.

ii) **Rise in Demand for Resource Flexibility and Utilization**

Disaggregated Storage Solutions and Shifting Performance Challenges: The escalating demand for fine-grained resource flexibility and optimal utilization and the deployment of disaggregated storage solutions have shifted performance challenges from hardware to software domains. Traditional remote storage access methods, such as iSCSI and lightweight Linux-based servers, struggle with unsustainable CPU overheads. This underscores the need for innovative solutions like i10 to address the evolving landscape of remote storage access effectively.

c) **Previous Works:**

i) **Performance Bottlenecks in Traditional Remote Storage Stacks:** Building upon insights from "Re-Flex: Remote Flash ≈ Local Flash" [Ana Klimovic, Heiner Litz, and Christos Kozyrakis, ACM ASPLOS, 2017], it is evident that traditional remote storage stacks encounter fundamental performance bottlenecks.

ii) **Limitations of iSCSI Protocol:** The iSCSI protocol [Satran et al., "Internet Small Computer Systems Interface (iSCSI)"], designed for remote HDD access over 1Gbps networks, achieves a modest ~70K IOPS per CPU core, highlighting its limitations in high-performance scenarios.

iii) **Performance of Lightweight Servers for Remote Storage Access:** Lightweight servers utilizing Linux libevent and libaio for remote storage access deliver approximately ~75K IOPS per core, as observed in "Re-Flex: Remote Flash ≈ Local Flash" [Ana Klimovic et al.].

iv) **Challenges with Distributed File Systems (HDFS and GFS):** While distributed file systems like HDFS and GFS excel in large data transfers, they prove less efficient for small-sized random read/write requests on high-throughput storage devices, as highlighted in the research mentioned earlier [Ana Klimovic et al.].

These observations underscore the performance limitations inherent in conventional remote storage solutions and set the foundation for exploring more effective approaches, such as the proposed i10 system.

d) **Design of i10:**

i) **Intermediary Layer Placement:**

1) i10 serves as an intermediary layer between the kernel storage stack (block device layer) and the kernel network stack. It facilitates the exchange of control and data plane messages through dedicated lanes known as i10 lanes, allocated at the (core, target) level.

ii) **i10-lanes and Request Aggregation:**

1) Dedicated i10-lanes process requests directed to the same target server.

2) Aggregation of multiple requests into "caravans" occurs, which are then transmitted over the same TCP connection. This aggregation minimizes network processing overhead, leveraging optimizations such as TSO (TCP Segmentation Offload) and GRO (Generic Receive Offload).

iii) **Delayed Doorbells for Context Switching Reduction:**

1) i10 introduces the concept of delayed doorbells to reduce context-switching overhead.

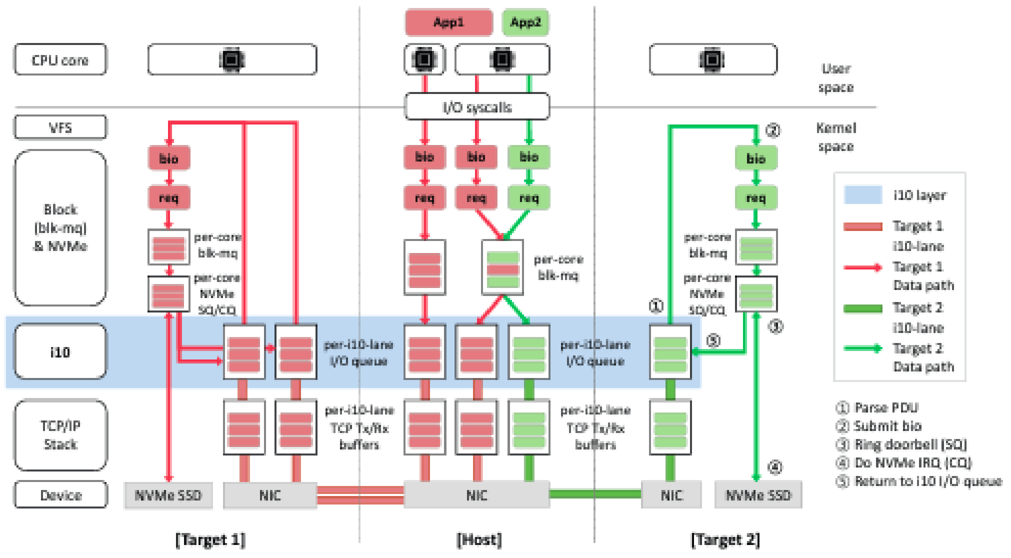
2) Instead of immediately signaling the storage device upon request creation, delayed doorbells allow accumulation, signaling the i10 worker thread later.

3) This significantly reduces context switching overhead and ensures appropriately sized caravans for transmission through the in-kernel socket interface.

iv) **Target-side Processing of Caravans:**

1) On the target side, i10 processes incoming caravans, regenerates requests, submits them to the block layer, and manages data flow back to the host.

2) The design involves i10-lanes, i10 caravans, and delayed doorbells to facilitate efficient remote storage access.



e) **Design Details:**

i) **i10 Lanes:**

1) Dedicated resources for each i10 lane include blk-mq Request Queues and optimizing request management.

2) i10 I/O Queues, resembling NVMe standard queues but communicating with remote target servers.

3) TCP Socket Instance: Maintains a long-lived TCP connection with the target server.

4) i10 Worker Thread: Dedicated for each core, responsible for bidirectional data movement, caravan aggregation, and efficient data processing.

ii) **i10 Caravan:**

   1) i10 combines multiple requests destined for the same destination into groups called "i10 caravans."

   2) Caravans are created at i10 queues to avoid the need for sorting requests at the block layer, minimizing CPU overhead caused by individual request processing.

   3) Each caravan can carry up to 64KB of data, with a limit on the number of requests to prevent overwhelming small requests.

iii) **Delayed Doorbells:**

   1) i10 introduces "delayed doorbells" to address high context-switching overhead over high-latency networks.

   2) Instead of immediate doorbell signaling, i10 waits until a specific number of requests accumulate, or a timeout occurs.

   3) This approach minimizes CPU overhead, reduces context switches, and enhances performance, especially under high loads.

f) Implementation:

i) **Kernel Integration and Optimization:** Seamless integration into the Linux kernel for compatibility with commodity hardware and unmodified applications. Optimized data transmission using kernel_sendmsg(), enhancing CPU usage and overall throughput.

ii) **Efficient Configuration and No-Delay Path:** "No-delay path" for latency-sensitive requests prioritizes immediate execution through flushing outstanding requests. Parameters like aggregation size (16) and doorbell timeout (50µs) are configured for optimal throughput-latency balance fixed 8MB TCP buffers, set explicitly at session establishment to prevent caravan fragmentation.

g) Benefits of i10:

i) **Seamless Integration and Compatibility:** Requires no application modifications, ensuring easy integration. Operates on the standard TCP/IP network stack, eliminating the need for network stack or hardware changes. Adheres to the NVMe-oF standard, ensuring compatibility with modern NVMe storage devices.

ii) **Efficient Performance:** Fully utilizes a 100Gbps link with standard server hardware, maintaining CPU efficiency comparable to modern solutions.

h) Evaluation Methodology:

i) **Single Core Performance:** i10 demonstrates superior throughput at high loads with slightly higher latency at low loads than NVMe-RDMA.

ii) **SSD access latency:** i10 (189µs) vs. NVMe-RDMA (105µs), with competitive RAM block device throughput.

iii) **Scalability with Cores:** i10 and NVMe-RDMA saturate SSDs with 4 cores, outperforming NVMe-TCP. i10 excels with RAM block devices, saturating a 100Gbps link with around 20 cores, surpassing NVMe-RDMA.

iv) **Performance with Read/Write Ratios:** i10 consistently outperforms NVMe-RDMA across all SSD workload variations. SSD scenarios are constrained by random write performance, while RAM block devices exhibit consistent throughput.

v) **Scalability with Multiple Targets:** i10's performance remains robust with increasing target devices, outperforming both NVMe-TCP and NVMe-RDMA. Saturates a 100Gbps link with 16 or more targets, showcasing low CPU contention and positive scalability trends.

i) **Results:**

i) **Design Contribution:** i10's design components, including i10 lanes, caravans, and delayed doorbells, play a pivotal role in performance. i10 lanes serve as a baseline, scaling effectively with increasing core counts. Optimizations like TSO/GRO and jumbo frames enhance throughput, particularly with RAM block devices. Caravans and delayed doorbells significantly contribute, boosting throughput by 38.2% and 23.2%, respectively, with 16 cores.

ii) **Bottleneck Understanding:** CPU profiling reveals i10's efficiency in reducing CPU cycles during network processing compared to NVMe-TCP. i10 addresses inefficient use of network capacity by generating larger packets, minimizing CPU usage. i10 minimizes context switching overhead, which is crucial for efficient operation.

iii) **CPU Efficiency:** i10 enhances CPU efficiency, providing applications with more CPU cycles. Compared to NVMe-TCP, applications gain 2.9× and 1.8× more CPU cycles with SSD and RAM devices, respectively. Outperforms NVMe-RDMA in providing more CPU resources for applications, particularly with RAM block devices.

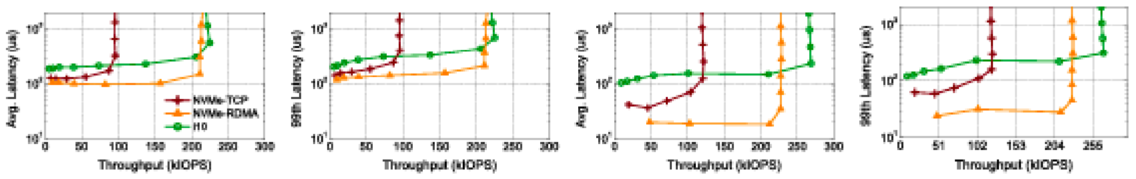
iv) **Shift in Bottlenecks:** i10 shifts performance bottlenecks from lower layers and network processing to the block device layer within the kernel. Highlights the potential for optimizing the block layer to enhance overall remote and local storage access performance.

v) **i10 Performance with RocksDB:** In RocksDB evaluation, i10 performs comparably to NVMe-RDMA with nearly a 1.2× improvement over NVMe-TCP in execution time. Ensures more CPU resources for the application, resulting in better overall performance.

j) **Future Enhancements:**

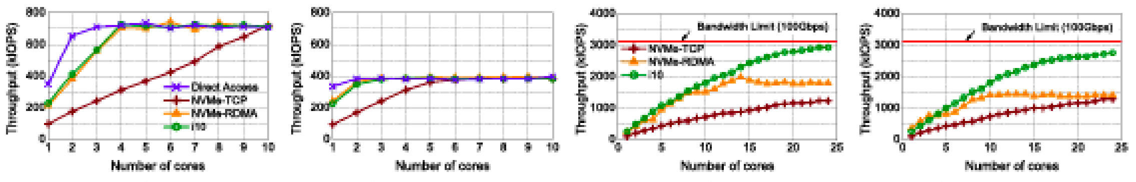
i) **Enhancing iSCSI Performance:** i10's optimization techniques promise to improve iSCSI performance by addressing suboptimal CPU usage during I/O transmission. Adapting i10 optimizations to iSCSI could reduce CPU overhead and enhance efficiency.

ii) **Integration with Emerging Transport Designs:** Future work can explore integration with emerging transport designs utilizing hardware offload to improve CPU efficiency. Combining i10's optimizations with cutting-edge transport designs can further elevate remote storage access performance.



(a) Average Latency (SSD) (b) Tail Latency (SSD) (c) Average Latency (RAM) (d) Tail Latency (RAM)

Figure 6: **Single core performance (4KB random read)**: when compared to NVMe-TCP, i10 achieves significantly higher throughput-per-core with comparable latency; when compared to NVMe-RDMA, i10 achieves comparable throughput while achieving average and tail latency within 1.4× and 1.7×, respectively, for the case of SSDs.



(a) Read (SSD) (b) Write (SSD) (c) Read (RAM) (d) Write (RAM)

Figure 7: **Multi-core performance (4KB random read/write)**: i10 achieves throughput significantly better than NVMe-TCP and comparable to NVMe-RDMA while operating on commodity hardware; i10 and NVMe-RDMA also achieve near-perfect scalability with number of cores.

Enhancing Network Interface Card (NIC) Performance with Loom

a) **General Introduction:**

   i) In contemporary data center environments, the efficient enforcement of hierarchical network policies poses a significant challenge due to the limitations inherent in existing multi-queue Network Interface Cards (NICs).

   ii) The prevailing issue revolves around the fact that current NIC architectures empower the NIC to make packet scheduling decisions, thereby limiting the flexibility and control that operating systems (OSes) can exert.

   iii) The inflexible nature of on-NIC packet schedulers further compounds the problem, making achieving effective traffic isolation among applications with diverse network objectives challenging.

b) **Motivation:**

   i) **Inadequate Application Isolation:** Current NIC architectures lack the flexibility to ensure effective isolation among applications with competing network objectives, particularly in scenarios involving bandwidth and latency-sensitive applications.

   ii) **Inflexibility in Packet Scheduling:** The NIC's packet scheduler exhibits inflexibility, evident when applying rate limiters to a 16-threaded Memcached application. Attempts to enforce rate limits on individual queues lead to overutilization or underutilization, violating network policies and fairness objectives.

   iii) **Challenges in Network Policy Enforcement:** Hybrid approaches, integrating Data Center Bridging (DCB) and software, face challenges in achieving fair bandwidth sharing. MQ-Priority struggles to ensure fair sharing, while MQ-Fair introduces high latency, highlighting the complexity of enforcing policies across multiple queues.

   iv) **Insufficient OS/NIC Interface:** The dynamic enforcement of network policies requires frequent updates to scheduling metadata, leading to performance issues. Current NIC interfaces necessitate separate PCIe writes for each update, causing delays and potentially impacting CPU availability.

   v) **Limitations in DAG Policies:** although effective at a local level, DAG policies encounter limitations in expressing comprehensive network-wide policies. The inability to guarantee algorithm efficiency on real-world hardware further complicates the effective implementation of policies.

c) **Past Work:**

   i) **SENIC:**

      1) SENIC is related to Loom by allocating a transmit descriptor queue for each flow and utilizing a doorbell FIFO for NIC notifications.

      2) SENIC faces potential performance bottlenecks due to the use of a single shared doorbell FIFO.

      3) SENIC lacks support for programmable policies.

   ii) **ConnectX-4 (CX-4) NIC:**

      1) CX-4 NIC shares similarities with Loom by supporting numerous descriptor queues and using per-CPU registers for doorbell updates.

      2) CX-4 faces potential performance issues as it requires sequential updates for individual doorbells and configurations.

      3) CX-4 does not support hierarchical or programmable policies.

d) **Loom's Proposal:**

   i) **Programmable Scheduling Leveraging Switch Advances:**

      1) Utilizes recent switch design advancements for programmable and stateful match+action forwarding pipelines.

      2) Implements programmable hierarchical packet scheduling for fair and efficient packet transmission.

   ii) **Expression of Scheduling Algorithms:**

      1) Defines scheduling algorithms in terms of rank/enqueue-dequeue functionalities.

      2) Maintains state across function calls to enable dynamic and adaptable scheduling.

   iii) **Challenge in Rate Limiter Addressal:**

      1) Addresses the limitation of the previous PIFO design using a single queue for both shaping and scheduling.

      2) Mitigates issues arising from the coexistence of rate-limited and non-rate-limited packets in the same queue.

   iv) **Programmable Scheduling in NIC with Limited Memory:**

      1) Overcomes small NIC memory space challenges by performing scheduling over packet descriptors.

      2) Leverages OS communication for scheduling metadata, assigning each flow its descriptor queue.

e) **Implementation:**

   i) **On-NIC Match+Action Pipeline:**

      1) Utilizes the on-NIC match+action pipeline for packet classification and scheduling.

      2) Implements lookup of scheduling metadata, including priority and destination information, for effective packet transmission.

   ii) **Descriptor Queue Enqueue in PIFO Tree:**

      1) Enqueues pointers to descriptor queues in a PIFO tree for scheduling competing packet transmissions.

      2) Computes scheduling ranks/priorities based on OS-provided information, determining packet priority in the PIFO tree.

   iii) **DMA Engine Operation:**

      1) Employs the DMA engine for data transfer from main memory to the NIC after dequeue.

      2) Reads descriptor and packet data from main memory and process packets through the match+action pipeline before transmission.

   iv) **Optimization for PCIe Writes:**

      1) Minimizes PCIe writes by using a batched doorbell mechanism for efficient notification of new segments.

2) Passes scheduling metadata updates in-line with packet descriptors to avoid additional PCIe writes.

v) Scheduling Metadata Communication to NIC:

1) Communicates scheduling metadata through doorbell descriptors for flow-wide information.

2) Uses segment descriptors for individual segment-specific scheduling metadata, providing flexibility with a trade-off in speed.

vi) Local SRAM for Dynamic Scheduling Hierarchy:

1) Stores configuration of each segment in local SRAM to facilitate dynamic reconfiguration of scheduling hierarchy.

2) Includes metadata and traffic class information in SRAM to avoid additional PCIe writes.

vii) DAG Rate Limiting for Efficient Global Shaping:

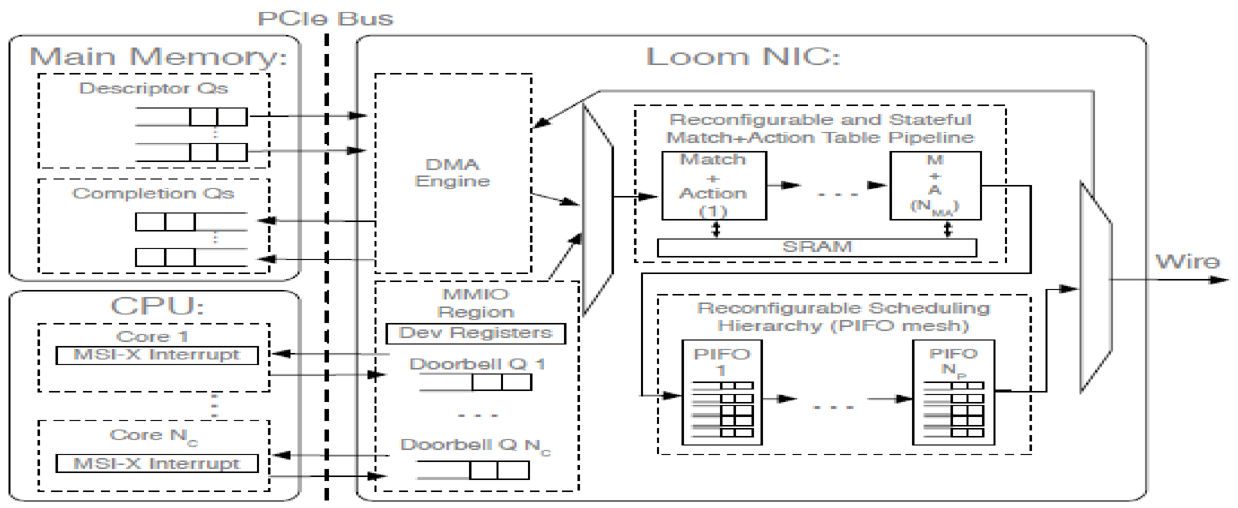
1) Introduces a Global Shaping queue to overcome limitations of per-node shaping queues.

2) Enqueues packets in the PIFO hierarchy based on priority and utilizes the rate limiter for efficient packet transmission.

viii) Line Rate Operation and Optimization:

1) Analyzes the worst-case scenario of 2 times enqueue/dequeue for a 100 Gbps NIC operating at 1 GHz.

2) Proposes optimization by including a flag in scheduling metadata for immediate enqueue of rate-limited packets and limiting outstanding packets per traffic class to optimize dequeue operations.



PROTOTYPE OF LOOM

f) Benefits:

i) Enhanced Flexibility:

1) Leverages recent advances in switch design for adaptable and programmable packet scheduling.

    2) Hierarchical scheduling supports a wide range of scheduling and shaping algorithms.

ii) **Dynamic Scheduling Algorithms:**

    1) Expresses algorithms in terms of rank/enqueue-dequeue functionalities for dynamic adaptation.
    2) Maintains state across functions, enabling responsiveness to changing network conditions.

iii) **Effective Rate Limiter Handling:**

    1) Mitigates issues related to rate limiting by separating rate-limited and non-rate-limited packets.
    2) Ensures smoother prioritization without conflicts in the scheduling process.

iv) **Optimized Scheduling in NIC with Limited Memory:**

    1) Efficiently schedules packets in the NIC's small memory space.
    2) Relies on OS communication for scheduling metadata, improving data transfer efficiency.

v) **Reduced PCIe Writes and Efficient Doorbell Notification:**

    1) Minimizes PCIe writes through a batched doorbell mechanism for streamlined processing.
    2) In-line transmission of scheduling metadata with packet descriptors reduces additional PCIe writes.

vi) **Flexible Communication of Scheduling Metadata:**

    1) Efficiently communicates flow-wide scheduling metadata through doorbell descriptors.
    2) Segment descriptors offer flexibility for individual segment-specific scheduling metadata.

vii) **Local SRAM for Dynamic Configuration:**

    1) Uses local SRAM for storing segment configuration, facilitating dynamic scheduling hierarchy changes.
    2) Reduces the need for additional PCIe writes by including metadata and traffic class information locally.

viii) **Efficient Global Shaping with DAG Rate Limiting:**

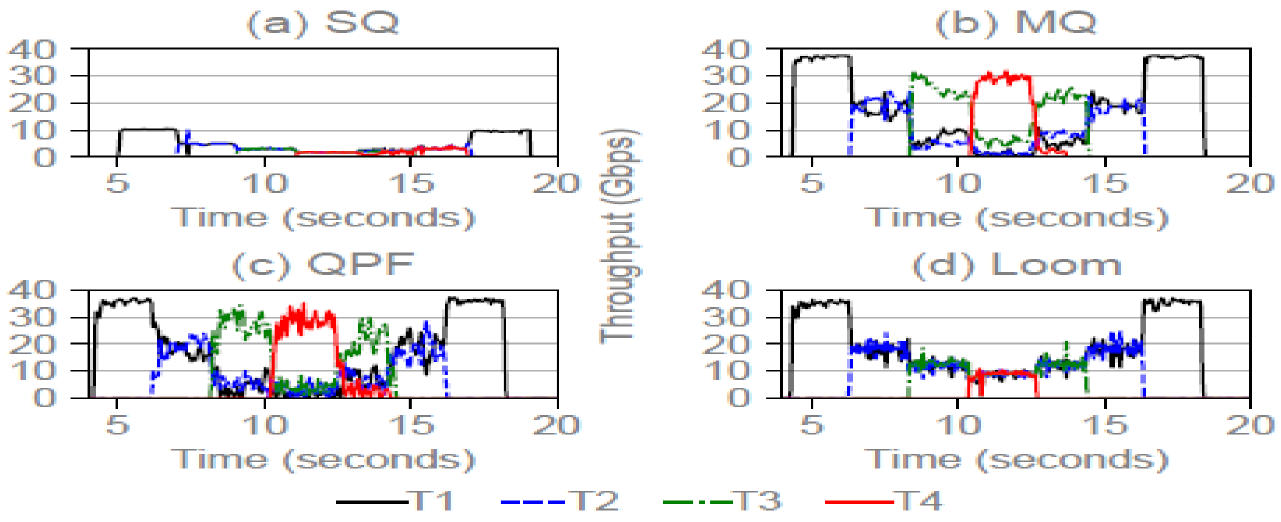
    1) Introduces Global Shaping queue for network-wide rate limiting, optimizing resource usage.
    2) Prioritizes packets based on priority in the PIFO hierarchy for effective shaping.

ix) **Optimal Line Rate Operation:**

    1) Demonstrates optimal line rate operation, ensuring efficient utilization of a 100 Gbps NIC operating at 1 GHz.

x) **Additional Optimization Strategies:**

    1) Proposes optimization strategies, like flag inclusion in scheduling metadata and limiting outstanding packets per traffic class, to enhance operational efficiency further.



g) **Evaluation Methodology:**

i) The experiment utilized a server configuration comprising Mellanox ConnectX-3 NICs, providing a 40 Gbps capacity. The server was equipped with an Intel E5-2650 8-core CPU (Haswell) and 64GB of RAM. Traffic generation involved sending packets with a 1500-byte MTU.

ii) Four Tenants, denoted as T1, T2, T3, and T4, were employed for the experiment. Each tenant operated its own flow without any imposed rate limitations.

h) **Results:**

i) Bandwidth was distributed fairly among tenants. All tenants experienced identical throughput levels simultaneously.

ii) The NIC efficiently transmitted an equal amount of traffic for all tenants concurrently. The redistribution of bandwidth occurred promptly.

iii) **Single Queue Method:** The total rate reached only 10Gbps, showcasing significant underutilization of the allocated 40Gbps resources.

iv) **Multiqueue and Queue per Flow Methods:**

1) The total rate approached 40Gbps; however, the introduction of a third tenant led to a prioritization shift, favoring the newcomer over T1 and T2.

2) A similar trend occurred with the arrival of the fourth tenant, resulting in a prioritization imbalance.

i) **Future Enhancements:**

i) **Dynamic Scheduling Configuration:**

1) Loom presently confines the configuration of the scheduling hierarchy to boot time only.

2) Introducing runtime support for scheduling hierarchy configuration would offer advantages to applications, eliminating the need for shutdowns during configuration adjustments.

ii) **Multi-NIC Support:** Loom is currently designed to support a single NIC. Extending support to multiple NICs would enable the construction of intricate load-balancing units and facilitate the implementation of advanced network features.